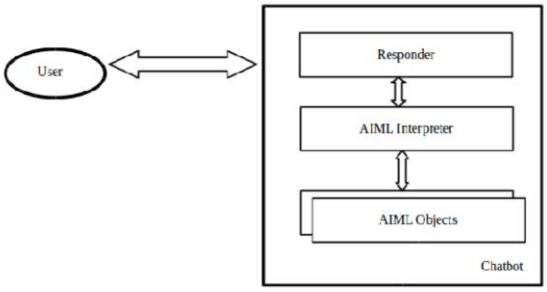
1. Artificial Intelligence Markup Language (AIML)

AIML is an extension of XML. The AIML data objects consist of two units: **Topics** and **Categories**. The purpose of the AIML chatbot is to facilitate a better conversation modeling. The **data object** in the AIML language has the responsibility of modeling the conversational patterns. The important objects are **categories**, **pattern**, and **template**. The category tag is used to specify the knowledge in the conversation. The pattern contains the input from the user to the system and the template includes the output or response to the user. The structure of the AIML categories, patterns, template are given below:

```
<category>
        <pattern> Input from the User </pattern>
                <template>
                        Response to the User
                </template>
</category>
```

The AIML chatbot is based on the **pattern matching concept**, the response generated is based on the mapping of keywords in each request and their patterns. With the help of the AIML Interpreter, the pattern matching between query and response is done.



1.1. Types of AIML Categories

There are three types of categories:

Atomic Categories: are those with patterns that do not have wildcard symbols, _ and *, for example:

```
<category>
        <pattern> 10 Dollars </pattern>
        <template> That is affordable </template>
</category>
```

In the above category, if the user inputs '10 Dollars', then the bot answers 'That is affordable'.

Default Categories: are those with patterns having **wildcard symbols** *or _. The wild-card symbols match any input but they differ in their alphabetical order. Assuming the previous input 10 Dollars, if the robot does not find the previous category with an atomic pattern, then it will try to find a category with a **default pattern** such as:

```
<category>
        <pattern> 10 * </pattern>
        <template> It is ten </template>
</category>
```

So, the bot answers, ‘It is ten’.

Recursive Categories: are those with templates having <srai> and <sr> tags, which refer to **recursive reduction rules**. Recursive categories have many applications: symbolic reduction that reduces complex grammatical forms to simpler ones; divide and conquer that splits an input into two or more subparts, and combines the responses to each; and dealing with synonyms by mapping different ways of saying the same thing to the same reply.

- Symbolic Reduction:

```
<category>
         <pattern> DO YOU KNOW WHAT THE * IS </pattern>
         <template>
            <srai> What is <star/> </srai>
         </template>
</category>
```

In this example, <srai> is used to reduce the input to the simpler form “what is *”.

- Divide and Conquer:

```
<category>
         <pattern> YES* </pattern>
         <template>
            <srai> YES </srai>
            <sr/>
         </template>
</category>
```

The input is partitioned into two parts, “yes” and the second part; *is matched with the <sr/> tag. <sr/>=<srai><star/></srai>

- Synonyms:

```
<category>
         <pattern> HALO </pattern>
         <template>
            <srai> Hello </srai>
         </template>
</category>
```

The input is mapped to another form, which has the same meaning.

1.2 Extra AIML Functionality

- Remembering values of a variable.

```
<category>
      <pattern>MY NAME IS *</pattern>
```

```
      <template>
         <set name="name"><star/></set> is a nice name.
      </template>
</category>

<category>
  <pattern>WHAT IS MY NAME</pattern>
  <template>
     Your name is <get name="name"/>.
  </template>
</category>
```

The bot remembers the inputted name in this example and can respond whenever asked.

- Providing a set of responses to one question.

```
<category>
    <pattern>Once I *</pattern>
    <template>
        <random>
            <li>Go on.</li>
            <li>Can you be more specific?</li>
            <li>I did not know that.</li>
            <li>Are you telling the truth?</li>
            <li>I don't know what that means.</li>
            <li>Try to tell me that another way.</li>
            <li>What is it?</li>
        </random>
    </template>
</category>
```

When any question beginning with ‘Once I’ is asked, a random response is provided from the list of inputs given in the dataset.

- Setting a category.

```
<category>
  <pattern>SCHOOL</pattern>
  <template>School is an important institution.</template>
</category>

<category>
  <pattern>_ SCHOOL</pattern>
  <template>
     <srai>SCHOOL</srai>  #this will call the upper school pattern
  </template>
</category>

<category>
  <pattern>SCHOOL *</pattern>
  <template>
```

```
      <srai>SCHOOL</srai>
  </template>
</category>

<category>
  <pattern>_ SCHOOL *</pattern>
  <template>
     <srai>SCHOOL</srai>
  </template>
</category>
```

The following categories will refer to the first category that was initialized.

- Indicating that the current pattern depends on a previous chatbot output.

```
<aiml version="1.0">
<topic name="the topic">
<category>
  <pattern>PATTERN</pattern>
  <that>THAT</that>
  <template>Template</template>
</category>
          ..
          ..
</topic>
</aiml>
```

The `<that>` tag is used for this purpose and is generally optional.

1.3. AIML Pattern Matching Algorithm

Before the matching process starts, a **normalization process** is applied for each input, to remove all punctuation; the input is **split** into two or more sentences if appropriate; and converted to uppercase. For example, if input is: "I do not know. Do you, or will you, have a robots.txt file?" Then after the normalization it will be: "DO YOU OR WILLYOU HAVE A ROBOTS DOT TXT FILE".

After the normalization, the AIML interpreter tries to **match word by word** to obtain the longest pattern match, as we expect this normally to be the best one. This behavior can be described in terms of the **Graphmaster** set of files and directories, which has a set of nodes called **nodemappers** and branches representing the first words of all patterns and wildcard symbols.

Assume the user input starts with word `X` and the root of this tree structure is a folder of the file system that contains all patterns and templates, the pattern matching algorithm uses **depth first search** techniques:

1) If the folder has a subfolder that starts with underscore then turn to "_/" and scan through it to match all words suffixed `X`. If no match is found, then proceed to step 2.

2) Go back to folder, try to find a subfolder that starts with word X. If so, turn to "X/" and scan for matching the tail of X. Patterns are matched. If no match is found, then proceed to step 3.

3) Go back to the folder, try to find a subfolder that starts with star notation. If so, turn to "*/" and try all remaining suffixes of input following "X" to see if one matches. If no match was found, change the directory back to the parent of this folder, and put "X" back on the head of the input.

When a match is found, the process stops, and the template that belongs to that category is processed by the interpreter to construct the output.

1.4. AIML for the Tamil Language:

AIML is language independent, so the Tamil chatbot can be developed based on the AIML tags.

```
<aiml version="1.0">
      <category>
        <pattern>வணக்கம் *</pattern>
       <template>நீண்ட நாள் செழிப்புடன் வாழ்</template>
      </category>
      </aiml>
```

Thus, when we greet the bot in Tamil, the bot responds in Tamil.

2. Creating the AIML Chatbot

Four steps are involved in creating the Chatbot:

2.1. Gathering the Dataset

The files that are used for training are stored as ".aiml" files. This is where all the AIML tags and conversation data is stored. In our case, we will be using two files that are stored in the data folder:

- **greetings.aiml** (Contains data for basic conversation)
- **cmchs.aiml** (Contains data regarding the details of the scheme)

The following screenshots show the contents of the files:

2.2. Training the Chatbot

For training the chatbot, we use the AIML Parser, which processes the dataset provided (details on how it processes the data has been discussed in 3.3.1.3) and saves the model as a **brain file**.

The following code is used to load the two files we created earlier:

```
<aiml version="1.0">
    <category>
        <pattern>LOAD AIML</pattern>
        <template>
            <!-- Load standard AIML set -->
            <learn>data/*.aiml</learn>
        </template>
    </category>
</aiml>
```

Then, we used the following code to create our model:

```
import os
import aiml

k = aiml.Kernel()

print("Parsing aiml files")
k.bootstrap(learnFiles="./pretrained_model/learningFileList.aiml",
            commands="load aiml")
print("Saving brain file: " + BRAIN_FILE)
k.saveBrain(BRAIN_FILE)
```

In order to train the model, we need to initialize a kernel and bootstrap the learning files to it. Then, the AIML parser runs and creates a file called **`aiml_pretrained_model.dump`** which is the brain file. The **`learningFileList.aiml`** file specifies the dataset to load into the kernel (The code shown at the top).

2.3. Predicting the Responses

Once we have created the model, we can now use the model to get our responses. For this, we require a Graphical User Interface (GUI). While we used Tkinter for the English version of the chatbot, in this case, we explore the integration of the chatbot into the web. Hence, flask was used to deploy a web server, and get and post the questions and responses respectively.

```
from flask import Flask, render_template, request
import os
import aiml

app = Flask(__name__)

BRAIN_FILE="./pretrained_model/aiml_pretrained_model.dump"
k = aiml.Kernel()

if os.path.exists(BRAIN_FILE):
```

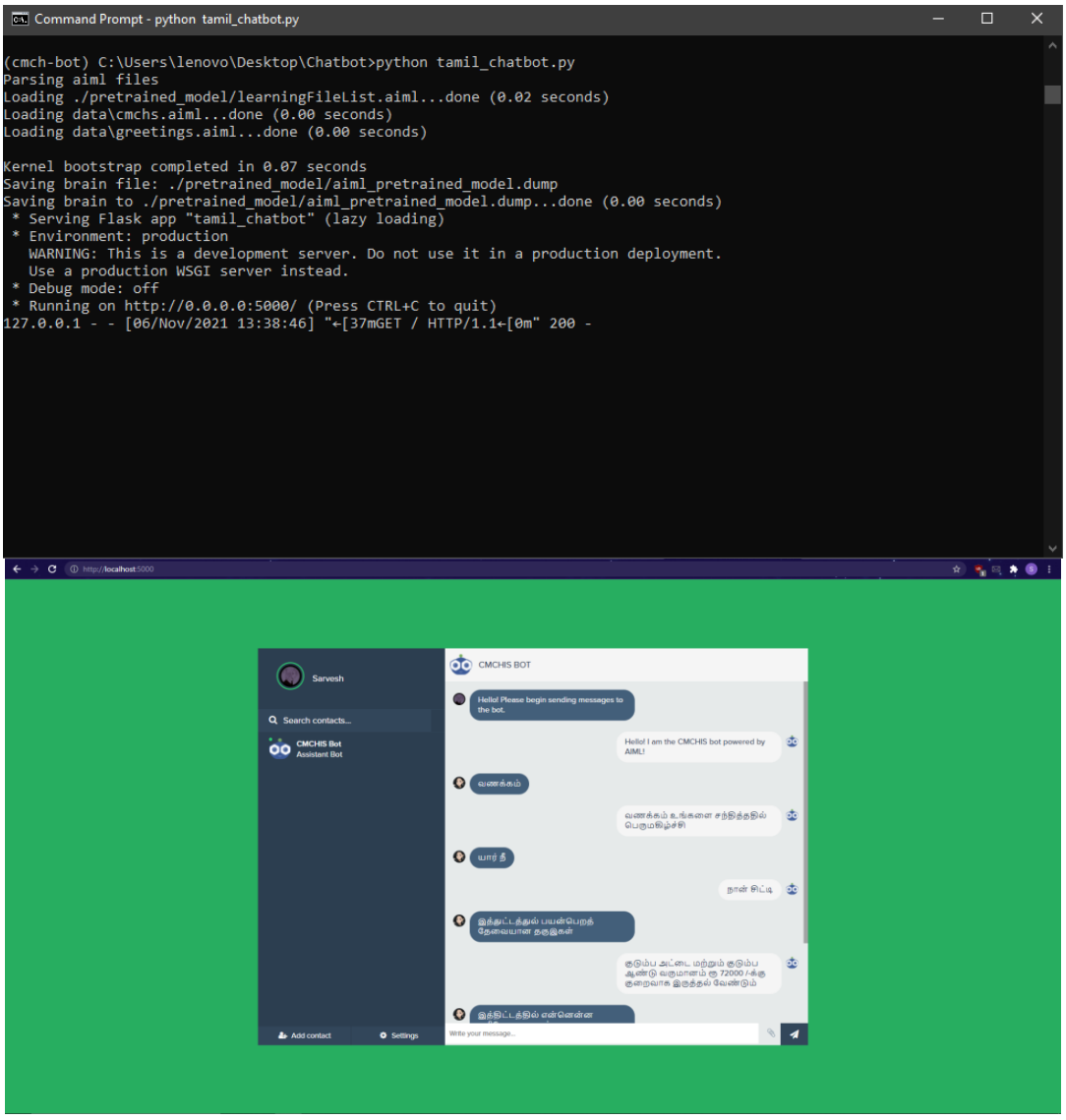
```
        print("Loading from brain file: " + BRAIN_FILE)
        k.loadBrain(BRAIN_FILE)
    else:
        print("Parsing aiml files")
        k.bootstrap(learnFiles="./pretrained_model/learningFileList.aiml",
                    commands="load aiml")
        print("Saving brain file: " + BRAIN_FILE)
        k.saveBrain(BRAIN_FILE)


    @app.route("/")
    def home():
        return render_template("home.html")


    @app.route("/get")
    def get_bot_response():
        query = request.args.get('msg')
        query = [w for w in (query.split())]
        question = " ".join(query)
        response = k.respond(question)
        if response:
            return (str(response))
        else:
            return (str("மன்னிக்கவும், எனக்கு புரியவில்லை. மீண்டும் சொல்ல
                        முடியுமா?"))


    if __name__ == "__main__":
        # app.run()
        app.run(host='0.0.0.0', port='5000')
```

The code first checks if the brain file already exists. If it does, then it directly loads the model, otherwise it uses the AIML parser and trains the model. Then, the web server is set up, and the home.html file is loaded, which provides the GUI to converse with the bot. the **`get_bot_response()`** function is used to receive questions and send responses. If any question is not understood by the bot, it sends the default response which is hard-coded.

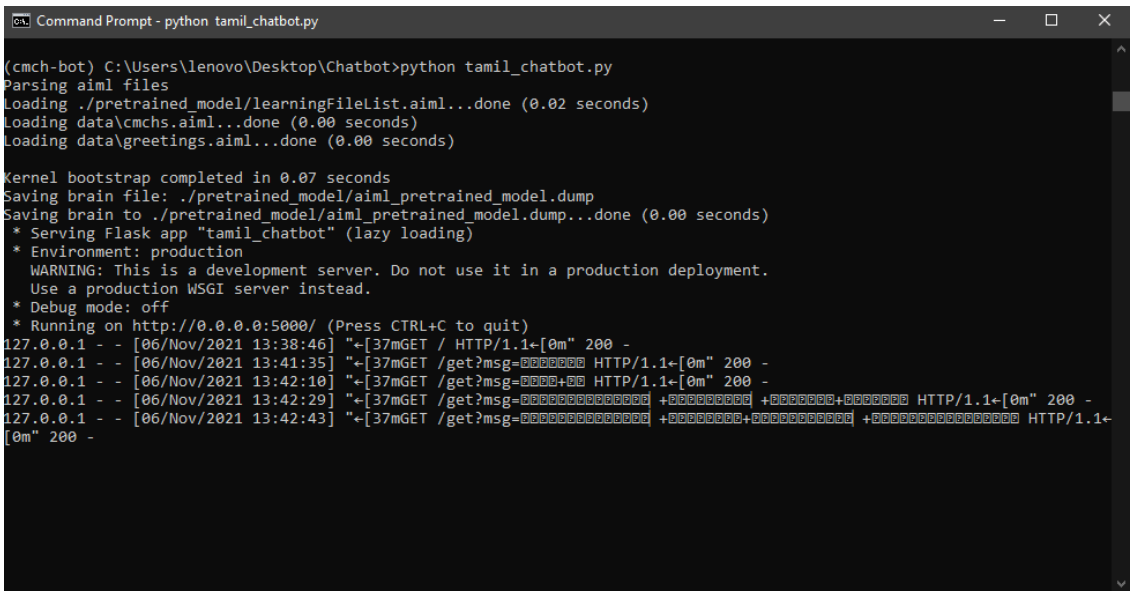
2.4. Running the Chatbot

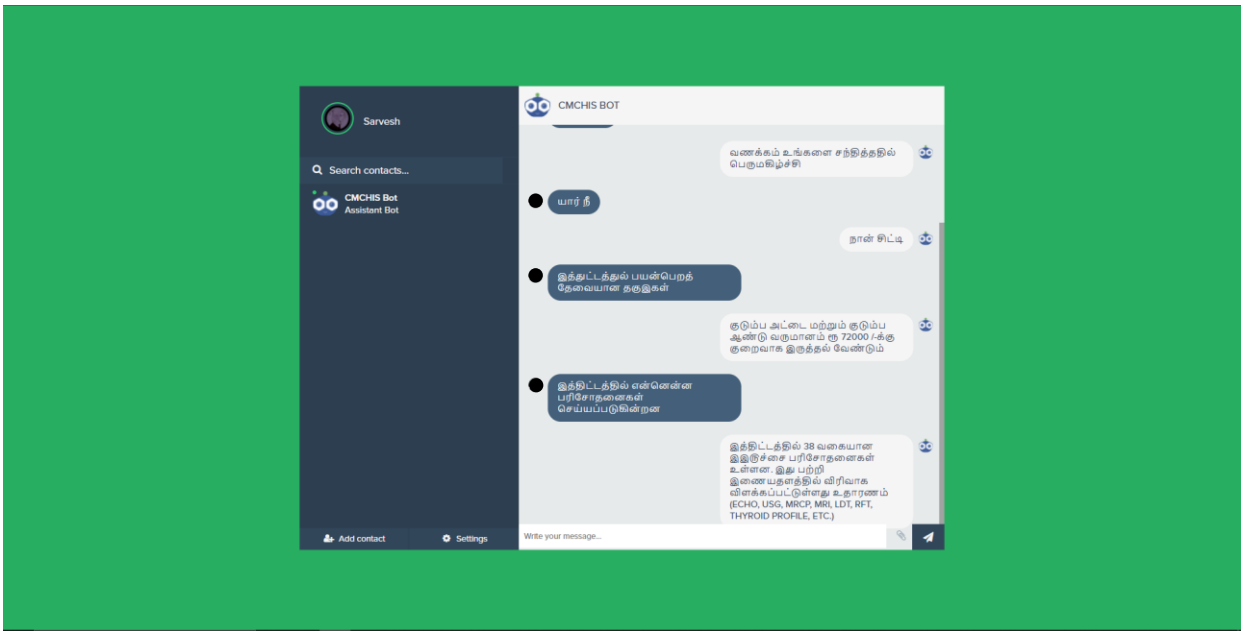
Once the above-mentioned files and directories are established, we run the **`tamil_chatbot.py`** file in the command terminal window which sets up the flask local server. Once that is done, we open a web browser of our choice and enter **`http://localhost:port_number`** (which will be mentioned in the command terminal) in the URL Address Bar to load the app.

The following screenshot shows the chatbot in action:



As we can observe, the chatbot responds accurately to our queries in the Tamil Language. The following screenshot shows the terminal window after our queries:





Thus, we observe that we get our responses fairly quickly and the bot is fairly accurate in matching the question patterns to their appropriate responses.

****